Object-Oriented Design and Programming

Object and Class

The two most important concepts in Object-Oriented Design (OOD) and Object-Oriented Programming (OOP) are *Object* and *Class*.

Objects are the primary building-blocks of the program. Objects are a bundle consisting of *data*, (*instance variables*), together with *methods*. At any time, a subset of the collection of a particular object’s attribute values comprises its *state variables* (see below). Each object can be thought of as a distinct entity which maintains its state, the current values of its state variables (or, simply, the object’s state). The execution of the program consists of objects being created and destroyed and, while alive, going through a sequence of state changes in response to its methods being invoked. When an object’s method is invoked (by an object, possibly itself), the object is said to have received a message.

The concept of *Class* has several points of view. One important one is that of a *template* or *object factory*. Every object is an *instance* of a class. The class is responsible for providing both the specifications for the objects (i.e. defining each instance variable and each method) as well as the mechanism for creating and initializing the objects (*instantiation*). A class specifies an object’s data by enumerating a list of instance variables, declaring each variable’s type and name. The class specifies an object’s methods by giving the method name, arguments, and return type (i.e. the value, if any, that is returned by the method to the one invoking that method). An Object-Oriented language provides a library of standard classes from which objects may be *instantiated* (created). The programmer may also define classes, either from scratch, or by extending or modifying existing classes via inheritance (see below).

A distinct, but related, perspective of Class is that of *type*; Classes are the object-oriented embodiment of *abstract data types*. Instances of classes may be considered to be of that classes type, and this information used to enforce parameter passing, return values, etc. A more important use of Class as abstract data type is ensuring that an object will respond to a given method. If that method is declared in the class definition and if an object is declared to be of that class, then the given object will indeed respond to the method. Being essentially static information, this enforcement may be made at compile time, the preferred place to trap program errors.

Object Data

An object's data may be refined into *attributes*, *parts*, and *static associations*. Attributes consist of *parameters*, permanent values of the object, and *state variables*, which may change value. Parts are references to other objects which only exist within the given object. Static associations are references to objects outside the object with whom the object will interact; the interactions take place when the object invokes one or more of the associated object's methods.

Encapsulation

Encapsulation (or *information-hiding*) is the rule that an object's data may only be changed by its methods. Each object is divided into its private part, which includes all its data and may have some methods, and its *public interface*. Other objects may therefore only interact with the given object by means of its public interface. The private part and public interface of an object is defined by its class.

Encapsulation is one of the primary means by which OOP supports code reusability and component-level modeling. Forbidding the direct manipulation of objects' data means objects must interact and communicate only through each others' public interface. This in turn encourages the design of objects which make fewer assumptions about their environment (that is, the other objects with which inhabit the model). This tends to make the objects more modular: if an object communicates with another object via given methods, it is relatively easy to communicate with other objects which have the same methods, even if the programmer had no idea the new objects were to be included in the model. The internal data of the objects may bear little resemblance to each other, yet if their behaviors (methods) are consistent and valid, then their internal representation is of no interest to objects which interact with them. The net effect is that design decisions made with respect to internal representations of objects are more easily modified and/or enhanced with minimal impact on other parts of the model (i.e. the other interacting objects). Decisions about the public interface, on the other hand, have much more impact and require more careful design.

Conversely, models containing objects that violate Encapsulation (i.e. directly manipulate the data of other objects) become difficult to extend and reuse. The internal representation of each object is specified by existing objects to a much greater extent. Changes to that internal data structure may necessitate drastic alterations of existing code, but not because the latter is defective in any way. It is exceedingly difficult to merge existing classes and objects which violate encapsulation, since Objects which are independently designed will have different internal structures.

Encapsulation is supported by all Object-Oriented languages, but it is typically not *mandatory*. It is therefore the responsibility of a program's designers and implementers to enforce. One device is the use of programming conventions. For example, it could be declared that all instance variables be declared “private,” that is, are inaccessible to objects outside the class in which they are declared.

To summarize, Encapsulation is the separation of an object’s public interface from its internal implementation; users and peers of the object interact only with the interface and should have no knowledge or dependence on the internal data and implementation.

Inheritance

Inheritance is the ability of a class to have all the data and methods of another class, possibly adding additional data and methods. The two classes are in a *subclass-superclass* relationship. Each instance of the subclass therefore has all the data and methods of the superclass as well as the new ones defined by the subclass. Classes may thus be organized into a *class hierarchy*, which can be represented as a rooted tree, with the root being the superclass of all classes.

A class declared to be a subclass of another is automatically endowed with all the methods and data of the superclass. Instances of the subclass respond to all the methods of the superclass. If none of the superclass’s methods are overridden (see below), then instances of the subclass should exhibit behavior that is indistinguishable from instances of the superclass. Internally, however, an instance of the subclass may be different. For example, data of the superclass may not be directly accessible to instances of the subclass. In that manner, an instance of the subclass may be treated as if it were simply another object. If the subclass adds new methods, then those methods may not necessarily be able to directly manipulate inherited data.

A frequent use of inheritance is for behavior refinement. An existing class has a desired behavior, but has implemented it in a manner that is not completely desirable. A subclass is created and the method is re-written (i.e., *overridden*; see below).

Another frequent use of inheritance is for implementing abstract methods (see below). Abstract methods are declared but not implemented. They are employed, for example, when it is desired for a given class and its subclasses to have a method of a particular name, but it is known that different subclasses may implement that method differently. Declaring the abstract method insures the existence of the behavior (i.e. that all subclasses will in fact respond to messages invoking that method) while allowing each subclass to implement the method as required by its specifications.

There are two components to inheritance: the interface specification (i.e., the names and signatures of the methods are obtained from the superclass) and the implementation specification (i.e., for non-abstract classes the implementation is obtained from the superclass). Groups of related method names may be represented as a “pure abstract class,” which is also sometimes called an *interface*. These interfaces may be used to specify the fact that a given class will have implemented all the listed methods. These interfaces may also be used as type information for ensuring that objects respond to given methods.

Multiple Inheritance is when a subclass has more than one immediate superclass. The subclass inherits all the methods and data from all its superclasses. This design technique may be useful in situations where more than one existing class has desired behaviors implemented. One difficulty with multiple inheritance is the potential for name ambiguity. Methods and data from different superclasses may have the same name, and it is the responsibility of the subclass to resolve the conflict. There are potentially undesirable design aspects to using multiple inheritance. One is a subtle violation of encapsulation: in order to resolve name and data conflicts, details of the implementation of the superclass must be known. Ideally, the public interface should be sufficient to "use" a class, whether by subclassing or by instantiation.

One solution is to allow only single inheritance in implementation, but to allow multiple inheritance with interfaces. In this scheme, a subclass may only inherit the actual methods and data of one class, but may be specified as conforming to as many interfaces as desired.

Polymorphism

In OOP, *polymorphism* typically refers to the ability to have more than one method of the same name. It can take a number of different forms.

One meaning of polymorphism is the fact that *different* classes may have methods of the same name. This is a weak form of polymorphism, since the methods having identical names may be coincidental. For example, both a random number generating class and a class that displays graphics could have a method called `draw()`. This is sometimes called *ad hoc polymorphism*. The utility of this type of polymorphism is more convenience than design. The program designer can use method names without fear of conflict with methods in other classes.

Method overloading is when the *same* class has multiple methods of the same name. Typically they are distinguished by their *signature*, or argument list. The compiler thus resolves a message utilizing that method by examining the signature. This is sometimes called *parametric polymorphism*, and need not be confined to OO languages. The benefits of overloading range from convenience to code robustness and reuse. Although it is not required, whenever possible all methods of a given name in a class should invoke one "master" method for the actual implementation. It is easier to verify the group of methods when designed in this manner. Furthermore, changes or extensions are easier to implement, since only the code in the master method needs to be changed.

Method Overriding occurs when a subclass redefines a method of its superclass. In this case the new method is identical to the overridden method in both name and signature. Instances of the subclass will respond to the new method, whereas objects which are instantiated from the superclass will respond to the overridden method. The compiler binds the object to the method depending on its class-determined type. If the subclass defines a method of the same name, but different signature, as the superclass, then the situation is more like method overloading. The subclass will have two methods with the same name but different signatures. Objects which interact with instances of the subclass won't be able to tell whether a given method was defined in the superclass or the subclass. Typically it is possible to explicitly invoke the superclass version of a method from the subclass's method. Overriding promotes code reuse in situations where an existing class is almost what is desired, but a few methods have undesirable implementations (i.e. unsuitable for the new use). The "bad" methods can simply be overridden, and the duplication of effort involved with re-implementing the remainder of the class is spared.

Abstract methods are methods which must be overridden in a subclass before objects may be instantiated. Classes containing abstract methods are *abstract classes*, and must be sub-classed before objects may be instantiated. In other words, there can be no instances of an abstract class. Abstract methods defer the responsibility of implementing methods that are designed to be present in the class (and subclasses), yet are known to be implemented differently in intended subclasses. Abstract methods enable a higher-level design, since they may be used as a placeholder for desired behavior. The implementation details for the desired behavior may vary with the different subclasses envisioned, or the implementation details may not even be known yet. The designer may simply want to defer the implementation in order to try out different algorithms for the method.

References

Booch, Grady. 1994. *Object-Oriented Analysis and Design, with Applications, Second Edition*. Addison-Wesley, Menlo Park, CA.

Budd, Timothy. 1997. *An Introduction to Object-Oriented Programming, Second Edition*. Addison-Wesley, Menlo Park, CA.

Cox, Brad and Andrew Novobilski. 1991. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Menlo Park, CA.